Practice Workbook: MongoDB Fundamentals

Pre-requisites

- To go through the following workbook you need to install MongoDB locally:
    - Windows Users [Link]
    - Mac Users [Link]
    - Linux Users [Link]

  Or alternatively:
    - Get started with MongoDB Atlas [Link]

1. Connecting to the Terminal

We connect to the Mongo terminal using the `mongo` command.

```
mongo
```

2. Creating and Dropping a database

We can switch to a database in Mongo with the use command.

```
use petshop
```

This will switch to writing to the `petshop` database. It doesn't matter if the database doesn't exist yet. It will be brought into existence when you first write a document to it.

You can find which database you are using simply by typing db. You can drop the current database and everything in it using `db.dropDatabase`.

Challenge

- Use the use command to connect to a new database (If it doesn't exist, Mongo will create it when you write to it).

3. Collections

Collections are sets of (usually) related documents. Your database can have as many collections as you like.

Because Mongo has no joins, a Mongo query can pull data from only one collection at a time. You will need to take this into account when deciding how to arrange your data.

You can create a collection using the createCollection command.

```
use petshop
db.createCollection('mammals')
```

Collections will also be created automatically. If you write a document to a collection that doesn't exist that collection will be brought into being for you.

View your databases and collections using the show command, like this:

```
show dbs
show collections
```

Challenge

- Use db.createCollection to create a collection.
- Run show dbs and show collections to view your database and collections.

4. Documents

Documents are JSON objects that live inside a collection. They can be any valid JSON format, with the caveat that they can't contain functions.

The size limit for a document is 16Mb which is more than ample for most use cases.

You can create a document by inserting it into a collection.

```
db.mammals.insert({name: "Polar Bear"})
db.mammals.insert({name: "Star Nosed Mole"})
```

Challenge

- Insert a couple of documents into your collection.

5. Finding Documents

Mongo comes with a set of convenience functions for performing common operations. Find is one such function. It allows us to find documents by providing a partial match, or an expression.

What can you do with Find?

You **can** use find to:

- Find a document by id
- Find a user by email
- Find a list of all users with the same first name
- Find all cats who are more than 12 years old
- Find all gerbils called 'Herbie' who are bald, have three or more eyes, and who have exactly 3 legs.

Limitations of Find

You **can't** use find to chain complex operators. You can do a few simple things like counting, but if you want to real power you need the *aggregate pipeline*, which is actually not at all scary and is quite easy to use.

The **Aggregate pipeline** allows us to chain operations together and pipe a set of documents from one operation to the next.

a. Using find

You can use find with no arguments to list documents in a collection.

```
db.entrycodes.find()
```

This will list all of the codes, 20 at a time.

You can get the same result by passing an empty object, like so:

```
db.entrycodes.find({})
```

b. Finding by ID

Assuming you know the object ID of a document. You can pull that document by id like so:

```
db.entrycodes.find(ObjectId("557afc91c0b20703009f7edf"))
```

The `_id` field of any collection is automatically indexed.

IDs are 12 byte `BSON` objects, not Strings which is why we need the `ObjectId` function. If you want to read more on ObjectId, you can do so here.

c. Finding by partial match

Say you have a list of users and you want to find by name, you might do:

```
db.people.find({name: "dave"})
```

You can match on more than one field:

```
db.people.find({
  name: "dave",
  email: "davey@aol.com"
})
```

You can match on numbers:

```
db.people.find({
  name: "dave",
  age: 69,
  email: "davey@aol.com"
})
```

You also match using a regex (although be aware this is slow on large data sets):

```
db.people.find({
  name: /dave/
})
```

Challenge

We need to start out by inserting some data which we can work with.

- Paste the following into your terminal to create a petshop with some pets in it

```
use petshop
db.pets.insert({name: "Mikey", species: "Gerbil"})
db.pets.insert({name: "Davey Bungooligan", species: "Piranha"})
db.pets.insert({name: "Suzy B", species: "Cat"})
db.pets.insert({name: "Mikey", species: "Hotdog"})
db.pets.insert({name: "Terrence", species: "Sausagedog"})
db.pets.insert({name: "Philomena Jones", species: "Cat"})
```

- Add another piranha, and a naked mole rat called Henry.
- Use find to list all the pets. Find the ID of Mikey the Gerbil.
- Use find to find Mikey by id.
- Use find to find all the gerbils.
- Find all the creatures named Mikey.
- Find all the creatures named Mikey who are gerbils.
- Find all the creatures with the string "dog" in their species.

6. Finding with Expressions and comparison queries

We have seen how we can find elements by passing Mongo a partial match, like so:

```
db.people.find({name: 'Yolanda Sasquatch'})
```

We can also find using expressions. We define these using JSON, like so:

```
db.people.find({
  age: {
    $gt: 65
  }
})
```

We can use operators like this:

- $gt - Greater than
- $lt - Less than
- $exists - The field exists

See the full list here:

http://docs.mongodb.org/manual/reference/operator/query/

Challenge

Copy the following code into a Mongo terminal. It will create a collection of people, some of whom will have cats.

Optionally modify the code so that some people have piranhas, and some have dachshunds.

```
use people
(function() {
  var names = [
    'Yolanda',
    'Iska',
    'Malone',
    'Frank',
    'Foxton',
    'Pirate',
    'Poppelhoffen',
    'Elbow',
    'Fluffy',
    'Paphat'
  ]
  var randName = function() {
    var n = names.length;
    return [
      names[Math.floor(Math.random() * n)],
      names[Math.floor(Math.random() * n)]
    ].join(' ');
  }
  var randAge = function(n) {
    return Math.floor(Math.random() * n);
  }
  for (var i = 0; i < 1000; ++i) {
    var person = {
      name: randName(),
      age: randAge(100)
    }
    if (Math.random() > 0.8) {
      person.cat = {
        name: randName(),
        age: randAge(18)
      }
    }
    db.people.insert(person);
  };
})();
```

- Use find to get all the people who are exactly 99 years old
- Find all the people who are eligible for a bus pass (people older than 65)
- Find all the teenagers, greater than 12 and less than 20.

$exists

We can use `exists` to filter on the existence of non-existence of a field. We might find all the breakfasts with eggs:

```
db.breakfast.find({
  eggs: {
    $exists: true
  }
})
```

Challenge

- Find all the people with cats.
- Find all the pensioners with cats.
- Find all the teenagers with teenage cats.

$gt and $lt

We can use $gt and $lt to find documents that have fields which are greater than or less than a value:

```
db.breakfast.find({
  starRating: {
    $gt: 5
  }
})
```

Challenge

We are going to use some real data now. The stocks json file is a list of all stocks traded in the US in 2015. It's real data.

Download the stocks.json file from here: https://bit.ly/3fgeskn

We can import a JSON file from the command line using the mongoimport shell command.

Enter the following into a **terminal**. Don't enter this into the Mongo console or it won't work.

```
mongoimport --db stocks --collection stocks --file stocks.json
```

- Find all the stocks where the profit is over 0.5
- Find all the stocks with negative growth

7. Projection

Find takes a second parameter which allows you to whitelist fields to pass into the output document. We call this projection.

You can choose fields to pass though, like so:

```
{
  ham: 4,
  eggs: 2
}
{
  cheese: 6,
  lime: 0.5
}
db.breakfast.find({}, {
  eggs: true,
  lime: true
})
```

This will yield

```
{
  eggs: 2
},
{
  lime: 0.5
}
```

Challenge

- Use projection to format your array of people. We want only the names.
- Output just the names of the people who are 99 years old
- Output only the cats, like this:

```
{ "cat" : { "name" : "Fluffy Frank", "age" : 13 } }
```

When you output the cats, you will need to find only people who have cats, where cats $exists, or you will have gaps in your data.

Aggregation

We can do much more complex projection, even creating new fields based on expressions using the aggregate pipeline. More on this in a bit.

Excluding the id field

You will notice that the ID field is always passed through project by default. This is often desirable, but you may wish to hide it, perhaps to conceal your implementation, or to keep your communication over the wire tight.

You can do this easily by passing `_id`: false:

```
db.breakfast.find({}, {
  eggs: true,
  lime: true,
  _id: false
})
```

Challenge

- List the cats. Remove the ids from the output.

8. Count, Limit, Skip & Sort

We can chain some additional functions onto our find query to modify the output.

a. Count

Count will convert our result set into a number. We can use it in two ways. We can either chain it:

```
db.people.find({sharks: 3}).count()
```

or we can use it in place of find:

```
db.people.count({sharks: 3})
```

To count the people who have exactly three sharks.

Don't confuse it with `length()`. Length will convert to an array, then count the length of that array. This is inefficient.

Challenge

- Find out how many people there are in total.
- Using your collection of people, and $exists, tell me how many people have cats.
- Use $where to count how many people have cats which are older than them.

Further Reading

Count can be a slow operation on large data sets. For more on optimising count, you might like to read the following: https://www.compose.io/articles/counting-and-not-counting-with-mongodb/

b. Limit and Skip

Limit will allow us to limit the results in the output set. Skip will allow us to to offset the start. Between them they give us pagination.

For example

```
db.biscuits.find().limit(5)
```

will give us the first 5 biscuits. If we want the next 5 we can skip the first 5.

```
db.biscuits.find().limit(5).skip(5)
```

Challenge

- Show the first 5 people
- Show the next 5 people
- Show the names and ages of the oldest 5 pensioners with piranhas
- Show the names and ages of the youngest 5 teenagers with cats, where the cats have the word "Yolanda" in their name.

c. Sort

We can sort the results using the sort operator, like so:

```
db.spiders.find().sort({hairiness: 1})
```

This will sort the spiders in ascending order of hairiness. You can reverse the sort by passing -1.

```
db.spiders.find().sort({hairiness: -1})
```

This will get the most hairy spiders first.

We can sort by more than one field:

```
db.spiders.find().sort({
  hairiness: -1,
  scariness: -1
})
```

We might also sort by nested fields:

```
db.spiders.find().sort({
  'web.size': -1
})
```

will give the spiders with the largest webs.

Challenge

- Find the youngest 1 person with a cat and a piranha.
- Give me just the name of the youngest 1 person with a cat and a piranha.
- Give me the 5 oldest cats
- Give me the next 5 oldest cats

Use the stocks data you downloaded to answer:

- Find the top 10 most profitable stocks
- Add a projection, tell which sector the most profitable stocks are in.
- Which is the least profitable sector.
- Have a look at the data. Spend a few minutes deciding which stocks you would most like to invest in.

9. Interacting with Cursors

When we compose a query, Mongo gives us back a cursor object from which we can get the values.

When we called limit and sort in the last section, we were actually calling methods on the cursor that was returned by find.

If we save the cursor in a variable we can call more methods on it.

```
var people = db.people.find( );
```

We can iterate over the the cursor using a simple while loop. We can check if the cursor has a next value, and can call cursor.next to get the value out.

```
var people = db.people.find();
while (people.hasNext()) {
   print(tojson(people.next()));
}
```

a. Functional Loops - forEach and map

We can simplify the code above using functional style loops.

```
db.people.find().forEach(function(person) {
  print(person.name);
});
```

We also have access to map, which will return an array of values.

```
var array = db.people.find().map(function(person) {
  return person.name;
});
```

Cursor persistence

Cursors last for 10 minutes and are then garbage collected. This should be sufficient for most tasks. If you need a longer lasting cursor for some reason, you can create a long lasting cursor, however you should be aware that it will eventually go out of sync with the database.

If you want to know how to prevent this behaviour, see here: http://docs.mongodb.org/manual/core/cursors/#closure-of-inactive-cursors

Challenge

You can read all the cursor methods here:

http://docs.mongodb.org/manual/reference/method/js-cursor/

- Iterate over each of the people and output them.
- Change the find function to find only the people with cats.
- Iterate over each of the people, outputting just the cat name and age each time.
- Use Map to generate an array containing all of the cat names.

Using the stocks data answer the following:

- Sort the stocks by profit.
- Iterate over the cursor and output all of the stocks names and tickers in order of profit.

10. Inserting, Updating & Deleting

CRUD is a basic function of any database. Crud stands for:

- Create
- Read

- Update
- Delete

The four basic things that any data store needs to give us.

a. Creating

We create using the insert command, like this:

```
db.people.insert({name: "Tony Stark", occupation: "Billionaire, playboy, philantropist..."})
```

The JSON object will be created and saved.

Challenge

- Refresh your muscle memory. Create a new person now. Ensure that person has a shark.

b. Reading

We have many options for finding. We have already seen db.collection.find(). We can also use `db.collection.findOne()` which will return at most one result.

As we shall see soon, we also have the aggregate framework, and if we need maximum flexibility at the expense of a good deal of speed, we can also use map-reduce.

Challenge

- Refresh your muscle memory. Find the person who has a shark.
- Use `findOne` instead of find. This will return only one document.

c. Updating

We save using the `db.collection.save` function. We pass the function a JSON object that contains the modified object to save, including the `_id`. The item will be found and updated.

```
var p = db.people.findOne()
p.age = 999
db.people.save(p)
```

We can also find and update in a single step using the update function:

```
db.people.update({name: 'dave'}, {name:'brunhilde'})
```

Challenge

- A year has gone by. Write a loop that iterates over a cursor and makes everyone one year older.
    - Remember to make the cats older too. See if you can do both in the same loop.

d. Deleting

We can remove people en-mass.

```
people = db.people.remove({name:'Dave'})
```

Challenge

- It's time for a cull. Delete all the 50 year olds.
- We also heard there was some guy running round with a shark. That's a dangerous animal. Take him out, in fact take out anyone with a shark.

11. The Mongo Aggregation Framework

The Mongo Aggregation framework gives you a document query pipeline. You can pipe a collection into the top and transform it though a series of operations, eventually popping a result out the bottom (snigger).

For example, you might take a result set, filter it, group by a particular field, then sum values in a particular group. You could find the total population of Iowa given an array of postcodes. You could find all the coupons that were used on Monday, and then count them.

We can compose a pipeline as a set of JSON objects, then run the pipeline on a collection.

a. Empty pipeline

If you provide an empty pipeline, Mongo will return all the results in the collection:

```
db.entrycodes.aggregate()
```

Challenge

- Try out the aggregate pipeline now. Call aggregate on your people collection. You'll see the result is the same as if you called find.

b. Filtering the pipeline with $match

We can use the aggregation pipeline to filter a result set. This is more or less analogous to find, and is probably the most common thing we want to do.

Say we want to list only people who have cats (where cat is a sub-document), we would probably do something like this this:

```
db.people.find({
  cat:{
    $exists: true
  }
})
```

We can get the same result in the aggregation framework using $match, like so:

```
db.people.aggregate({
  '$match' : {
    cat:{
      $exists: true
    }
  }
})
```

So why use aggregation over find? In this example they are the same, but the power comes when we start to chain additional functions as we shall soon see.

Challenge

- Use the people dataset. Match all the people who are 10 years old who have ten year old cats.
- Match all the people who are over 80 years old, and who's cats are over 15 years old.

When to use match

Matching is quick but not smart. It's designed to limit the result set, so that the rest of the pipeline can run more quickly. When used with project we can match against fields that don't exist in our result set. This is a powerful and useful feature.

c. Modifying a stream with $project

The find function allowed us to do simple whitelist projection. The aggregate pipeline gives us many more options.

We can use `$project` to modify all the documents in the pipeline. We can remove elements, allow elements through, rename elements, and even create brand new elements based on expressions.

Say we have a set of voucher codes, like this:

```
{
  "firstName" : "Dave",
  "lastName" : "Jones",
  "code" : "74wy zphz",
  "usedAt" : ISODate("2015-06-13T17:47:20.423Z"),
  "email" : "123@gmail.com"
},
{
  "firstName" : "Stuart",
  "lastName" : "Hat",
  "code" : "7uwz e3cw",
  "usedAt" : ISODate("2015-06-13T17:47:50.489Z"),
  "email" : "456@gmail.com"
}
...
```

We can use project to restrict the fields that are passed through. We pick the fields we like and set true to pass them through unchanged.

```
db.entrycodes.aggregate(
  {
    '$project' : {
      email: true,
      code: true
    }
  }
)
```

d. Removing the id

Remove the `id` field bay passing `_id: false`.

This will yield a set something like the following:

```
[
  {
    "code" : "7uwy zphz",
    "email" : "123@gmail.com"
  },
  {
    "code" : "7uwz eccw",
    "email" : "123@gmail.com"
  }
]
```

e. Renaming Fields

We can use project to rename fields if we want to. We use an expression: `$lastName` to pull out the value from the lastName field, and project it forward into the surname field.

```
db.entrycodes.aggregate(
  {
    '$project' : {
      surname: "$lastName"
    }
  }
)
```

This will yield something like the following:

```
[
  {
    surname : "Jones"
  },
  {
    surname : "Hat"
  }
  ...
]
```

f. Chaining $match and $project

We can chain `$match` and `$project` together. Say we have a list of codes, and some have not yet been used. We want to pull out the names and emails, but only from the codes which have been used.

```
[,
  {
    "code" : "7uwz eccw"
  }
  {
    "firstName" : "Dave",
    "lastName" : "Jones",
    "code" : "7uwy zphz",
    "usedAt" : ISODate("2015-06-13T17:47:20.423Z"),
    "email" : "123@gmail.com",
    "winner": true
  },
  {
    "firstName" : "Stuart",
    "lastName" : "Hat",
    "code" : "7uwz eccw",
    "usedAt" : ISODate("2015-06-13T17:47:50.489Z"),
    "email" : "123@gmail.com"
  },
  {
    "code" : "7uwz eccw"
  }
  ...
]
```

We might first `$match` the codes which have a `usedAt` field, and then use `$project` to pull out the names and emails from the remainder.

```
db.entrycodes.aggregate(
  {
    $match: {
      usedAt: {
        $exists: true
      }
    }
  },
  {
    $project: {
      firstName: true,
      lastName: true,
      email: true,
      _id: false
    }
  }
)
```

Challenge

- Make a list of cat names.
- First $match people with cats, or the output will be a bit sparse.
- Now use $project to pull out only the cat names. You will need to use the dot syntax: '$cat.name'.

Use the stocks JSON file.

- Rename `"Profit Margin"` to simply `"Profit"`. Surpress all other output including the id. We only want to see profit, the company name and the ticker.

g. Creating new fields with $project

We can use project to add new fields to our documents based on expressions.

Say we had a list of people, like so:

```
{firstName: "Chinstrap", surname: "McGee"}
{firstName: "Bootle", surname: "Cheeserafter"}
{firstName: "Mangstrang", surname: "Fringlehoffen"}
```

We can use project to compose a name field, like so:

```
db.entrycodes.aggregate(
  {
    $project: {
      name: {$concat: ['$firstName', ' ', '$surname']}
    }
  }
)
```

This will give us results like this:

```
{ "name" : "Dave Smith" },
{ "name" : "Mike Moo" }
```

h. Conditional fields with $cond

We can set the value of a field using a boolean expression using `$cond`. There are a couple of ways to use $cond. You may wish to review the documentation here: http://docs.mongodb.org/manual/reference/operator/aggregation/cond/

Say we have a set of customers, and some of them have complaints. We might set a flag on all of the unhappy customers like so:

```
db.customers.aggregate(
  {
    $project:{
      unhappy: {
        $cond: { if: '$complaints', then: true, else: false }
      },
      complaints: '$complaints'
    }
  }
)
```

Unhappy will either be true or false.

Challenge

- Use projection to add a hasCat field. This might form the basis for a future grouping or counting.

Using the stocks data:

- Modify your stocks. Project the profit as a profit field.
- Add a `isProfitable` field to show if the profit is greater than 0.
- Add a `buyNow` field to show if the profit is greater than 0.5

i. Grouping with $group

`$group` allows us to group a collection according to criteria. We can group by fields that exist in the data, or we can group by expressions that create new fields.

Group will operate on a set of documents in the pipeline, and output a new set of documents out the bottom.

We group using the `_id` field. This will create a new `_id` for each group that will be an object containing the grouping criteria.

The simplest group would look like this:

```
db.people.aggregate(
  {
    $group: {
      _id: 1
    }
  }
)
```

The id field is empty, so the group contains the whole collection, but we haven't output anything, so each output document is empty.

Grouping by id

If we just want to group by a single field we can do this easily. The id of each output document will be the value of the expression, in this case '$name'.

```
db.people.aggregate(
  {
    $group: {
      _id: '$name'
    }
  }
)
```

Challenge

- Try this out on your people data set. You should get a list of distinct names.
- The output is untidy, each name output in the id field. Add a `$project` step to the pipeline to rename the `'_id'` field to 'name'.

You just wrote a function for getting distinct emails.

Grouping by multiple fields

You can group on more than one field by passing an object to _id:

```
db.people.aggregate(
  {
    $group: {
      _id: {
        name: '$name',
        age: '$age'
      }
    }
  }
)
```

Challenge

- Try out the above. Notice that the _id field is now an object. Use $project to reformat the data. You now have distinct names and ages.

$pushing data into the result

When grouping we use the _id field to hold the common values that we are grouping our documents on. This means that the output of a group aggregation only holds the data that is common to all documents in that group.

What happens though if we want to preserve a value that we are not grouping on. For this we use $push.

$push

$push will create an array and store part or all of the grouped source documents in it.

```
db.entrycodes.aggregate([
  {
    $group: {
      _id: "$email",
      count: {$sum: 1},
      entry: {
        $push: {
          firstName: "$firstName",
          lastName: "$lastName"
        }
      }
    }
  }
])
```

$$ROOT

The $$ROOT variable contains the source documents for the group. If you'd like to just pass them through unmodified, you can do this by $pushing $$ROOT into the output from the group.

```
db.entrycodes.aggregate([
  {
    $group: {
      _id: "$email",
      name: "$firstName"
      count: {$sum: 1},
      entries: { $push: "$$ROOT" }
    }
  }
])
```

Challenge

- Use $match to select only people with cats
- Now group by name, and for each person.
- Push the cats into the result.

We can now see a list of all the cats owned by people with a particular name.

j. Counting with Group

Group has the ability to count. You can count the entries in a group. By chaining $group commands together you could count the number of groups.

For example, say you have a set of customer records which may contain duplicate emails. You could group by email and find out who is using your service most often. You might count the groups, to get the number of distinct emails, you might group by count, to find how many people used your site once, twice, five times, etc.

We use $group to count, because generally we want to count groups.

Counting everything

We could count the entire collection by grouping everything, then adding a count field. This is the same as `db.collection.find().count()`

```
db.hamsters.aggregate(
  {
    $group: {
      _id: 1,
      count: {
        $sum: 1
      }
    }
  }
)
```

Challenge

- Count all the people. How many are there?
- Add a $match step to the start of your pipeline. Count all the people with cats using the aggregate pipeline. How many do you have?
- Use project to create a 'hasCat' field. You will need to use $cond to do this: http://docs.mongodb.org/manual/reference/operator/aggregation/cond/. Check that your pipeline now contains the hasCat field.

k. Sorting with Aggregation

We can sort records in the aggregation pipeline just as we can with find. Choose the fields to sort on and pass 1 or -1 to sort or reverse the sort.

```
db.people.aggregate(
  {
    $sort: {
      age: 1,
      name: 1
    }
  }
)
```

Challenge

- group the stocks data by sector.
- Use $sum to discover the most profitable sector
- Sort by profitability
- $project the results

Aggregation by date

Stats are cool. People love stats. We can use Mongo to generate a timeline showing day by day activity.

To do this we will need to group by date. First we add fields to our documents representing `year`, `month` and `dayOfMonth`. Then we group by these fields. We can count to get an aggregate, or filter based on some parameter.

For clarity, we have separated this into group and project stages so you can see how the pipeline changes. You could roll these two steps into a single $group stage:

```
db.competitionentries.aggregate(
  {
    $project: {
      year: {$year: date},
      month: {$month: date},
      dayOfMonth: {$dayOfMonth: date}
    }
  },
  {
    $group: {
      _id: {
        year: '$year',
        month: '$month',
        dayOfMonth: '$dayOfMonth'
      },
      count: {
        $sum: 1
      }
    }
  }
)
```

Challenge

- Download the holidays dataset from here: https://bit.ly/34MW9S7
- Now import it into Mongo using mongoimport, something like this:

```
mongoimport --db holidays --collection holidays --file holidays.json
```

- Group the data by year, month and dayOfMonth. Be aware that not every holiday has a date. You may generate the dynamic fields with $project, or directly in $group _id.
- $push the holiday name into the result set.
- Sort by year, month and dayOfMonth.

You now have a calendar of events for the year.